



下载APP



12 | 高性能网络实战（上）：如何开发一个负载均衡器？

2022-02-11 倪朋飞

《eBPF 核心技术与实战》

[课程介绍 >](#)**讲述：山荣**

时长 20:37 大小 18.89M



你好，我是倪朋飞。

上一讲，我带你一起梳理了 eBPF 的安全能力，并教你使用 eBPF 分析和阻止了容器进程的安全问题。在开篇词中我就提到，eBPF 的主要应用场景涵盖了故障诊断、性能监控、安全控制以及网络优化等。从 [07 讲](#) 进入实战进阶篇开始，我已经为你介绍了应用于前三个场景中的内核跟踪、用户态跟踪、网络跟踪以及安全控制等。那么，对于最后一个场景，网络性能优化，eBPF 是如何发挥作用的呢？

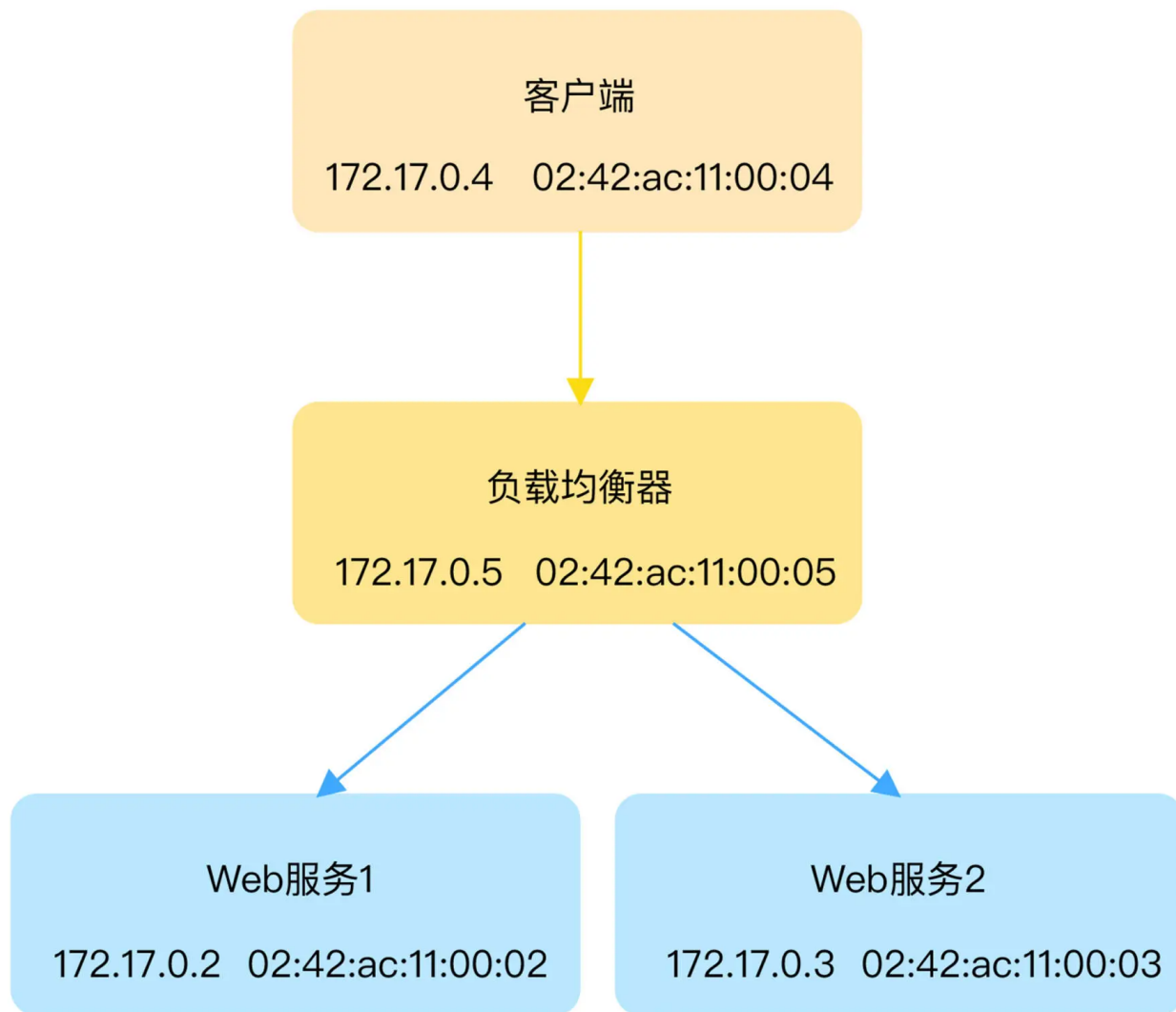
今天，我就以最常用的负载均衡器为例，带你一起来看看如何借助 eBPF 来优化网络的性能。



Nginx 负载均衡器

既然要优化负载均衡器的网络性能，那么首先就需要有一个优化的目标，即初始版的负载均衡器。在今天的案例中，我们使用最常用的反向代理和 Web 服务器 Nginx 作为初始版的负载均衡器，同时也使用自定义的 Nginx 作为后端的 Web 服务器。

为了方便环境的重现，负载均衡器、Web 服务器以及客户端都运行在容器中，它们的 IP 和 MAC 等基本信息如下图所示：



参考 Nginx 官方文档中 [HTTP 负载均衡](#) 的配置方法，你可以通过以下步骤来搭建上述的案例环境。

1) 执行下面的命令，创建上图中的 4 个容器：

```
1 # Webserver (响应是hostname, 如 http1 或 http2)
```

复制代码

```
2 docker run -itd --name=http1 --hostname=http1 feisky/webserver
3 docker run -itd --name=http2 --hostname=http2 feisky/webserver
4
5 # Client
6 docker run -itd --name=client alpine
7
8 # Nginx
9 docker run -itd --name=nginx nginx
```

注意，这儿启动的 Nginx 容器使用的还是官方镜像，还需要额外的步骤更新它的负载均衡配置。

小提示：在默认安装的 Docker 环境中，假如你没有运行其他容器，运行上述命令后得到的 IP 地址跟图中是相同的。

2) 执行下面的命令，查询两个 Web 服务器的 IP 地址：

[复制代码](#)

```
1 IP1=$(docker inspect http1 -f '{{range .NetworkSettings.Networks}}{{.IPAddress
2 IP2=$(docker inspect http2 -f '{{range .NetworkSettings.Networks}}{{.IPAddress
3 echo "Webserver1's IP: $IP1"
4 echo "Webserver2's IP: $IP2"
```

命令执行后，你将会看到如下的输出：

[复制代码](#)

```
1 Webserver1's IP: 172.17.0.2
2 Webserver2's IP: 172.17.0.3
```

3) 执行下面的命令，生成并更新 Nginx 配置：

[复制代码](#)

```
1 # 生成nginx.conf文件
2 cat>nginx.conf <<EOF
3 user  nginx;
4 worker_processes  auto;
5
6 error_log  /var/log/nginx/error.log notice;
7 pid       /var/run/nginx.pid;
```

```
8
9  events {
10     worker_connections 1024;
11 }
12
13 http {
14     include      /etc/nginx/mime.types;
15     default_type  application/octet-stream;
16
17     upstream webservers {
18         server $IP1;
19         server $IP2;
20     }
21
22     server {
23         listen 80;
24
25         location / {
26             proxy_pass http://webservers;
27         }
28     }
29 }
30 EOF
31
32 # 更新Nginx配置
33 docker cp nginx.conf nginx:/etc/nginx/nginx.conf
34 docker exec nginx nginx -s reload
```

配置完成后，再执行下面的命令，验证负载均衡器是不是生效了（ / # 表示在容器终端中执行命令）：

[复制代码](#)

```
1 # 查询Nginx容器IP (输出为172.17.0.5)
2 docker inspect nginx -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
3
4 # 进入client容器终端，安装curl之后访问Nginx
5 docker exec -it client sh
6
7 # (以下命令运行在client容器中)
8 / # apk add curl wrk --update
9 / # curl "http://172.17.0.5"
```

如果一切正常，多次执行 curl 命令后，你会看到如下的输出，即通过 Nginx 成功获得了两个 Web 服务器的输出，说明负载均衡器配置成功了：

 复制代码

```
1 / # curl "http://172.17.0.5"
2 Hostname: http1
3
4 / # curl "http://172.17.0.5"
5 Hostname: http2
```

负载均衡器配置成功后，它的性能怎么样呢？进入 client 容器终端中，执行下面的命令，就可以使用  **wrk** 给它做个性能测试：

 复制代码

```
1 / # apk add wrk --update
2 / # wrk -c100 "http://172.17.0.5"
```

稍等一会，你可以看到如下的性能测试报告：

 复制代码

```
1 Running 10s test @ http://172.17.0.5
2   2 threads and 100 connections
3   Thread Stats   Avg      Stdev     Max    +/-  Stdev
4   Latency       7.53ms    4.96ms   39.33ms   70.78%
5   Req/Sec       6.96k     514.59    8.88k    74.00%
6  138711 requests in 10.05s, 21.83MB read
7 Requests/sec:  13798.11
8 Transfer/sec:      2.17MB
```

从报告中你可以发现，默认情况下，总的平均每秒请求数是 13798，而每个线程的平均请求数和请求延迟是 6.96k 和 7.53 毫秒（在你的环境下可能看到不同数值，具体的性能指标取决于运行环境和配置）。你可以记录一下这些数值，以便后面跟 eBPF 进行比较。

如何使用 eBPF 优化负载均衡性能？

有了待优化的 eBPF 负载均衡器之后，接下来就是使用 eBPF 进行优化了。在开始接下来的内容之前，你可以先回顾一下我们之前课程的内容，思考有哪些可能的方案可以用在负载均衡的性能优化场景中。

在 [06 讲](#) 中我曾提到过，每个 eBPF 程序都属于特定的类型，不同类型 eBPF 程序的触发事件是不同的。既然是网络的性能优化，自然应该去考虑网络类的 eBPF 程序。根据触发事件的不同，网络类 eBPF 程序可以分为 XDP 程序、TC 程序、套接字程序以及 cgroup 程序。这几类程序的触发事件和常用场景分别为：

XDP 程序在网络驱动程序刚刚收到数据包的时候触发执行，支持卸载到网卡硬件，常用于防火墙和四层负载均衡；

TC 程序在网卡队列接收或发送的时候触发执行，运行在内核协议栈中，常用于流量控制；

套接字程序在套接字发生创建、修改、收发数据等变化的时候触发执行，运行在内核协议栈中，常用于过滤、观测或重定向套接字网络包。其中，

`BPF_PROG_TYPE_SOCKET_OPS`、`BPF_PROG_TYPE_SK_SKB`、`BPF_PROG_TYPE_SK_MSG` 等都可以用于套接字重定向；

cgroup 程序在 cgroup 内所有进程的套接字创建、修改选项、连接等情况下触发执行，常用于过滤和控制 cgroup 内多个进程的套接字。

根据这些触发事件，你可以发现这几类网络程序都有可能用在网络性能优化上。其中，由于支持卸载到硬件，XDP 的性能应该是最好的；而由于直接作用在套接字上，套接字程序和 cgroup 程序是最接近应用的。

既然有多种不同的性能优化方式，我就以套接字和 XDP 这两种方式为例，带你优化负载均衡的性能。由于内容比较多，接下来我们先看套接字 eBPF 程序的优化方法，而 XDP 方法我会在下一讲中为你介绍。

使用套接字 eBPF 程序优化网络性能

根据原理的不同，套接字 eBPF 程序又分为很多不同的类型。其中，`BPF_PROG_TYPE_SOCKET_OPS`、`BPF_PROG_TYPE_SK_SKB`、`BPF_PROG_TYPE_SK_MSG` 等类型的 eBPF 程序可以与套接字映射（如 `BPF_MAP_TYPE_SOCKETMAP` 或 `BPF_MAP_TYPE_SOCKHASH`）配合，实现套接字的转发。

套接字 eBPF 程序工作在内核空间中，无需把网络数据发送到用户空间就能完成转发。因此，我们可以先猜测，它应该是可以提升网络转发的性能（当然，具体能不能提升，还需要接下来的测试验证）。

具体来说，使用套接字映射转发网络包需要以下几个步骤：

1. 创建套接字映射；
2. 在 BPF_PROG_TYPE_SOCKET_OPS 类型的 eBPF 程序中，将新创建的套接字存入套接字映射中；
3. 在流解析类的 eBPF 程序（如 BPF_PROG_TYPE_SK_SKB 或 BPF_PROG_TYPE_SK_MSG）中，从套接字映射中提取套接字信息，并调用 BPF 辅助函数转发网络包；
4. 加载并挂载 eBPF 程序到套接字事件。

接下来，我们一起看看具体每一步该如何操作。

创建套接字映射

首先，第一步是创建一个套接字类型的映射。以 BPF_MAP_TYPE_SOCKHASH 类型的套接字映射为例，它的值总是套接字文件描述符，而键则需要我们去定义。比如，可以定义一个包含 IP 协议五元组的结构体，作为套接字映射的键类型：

 复制代码

```
1 struct sock_key
2 {
3     __u32 sip;    //源IP
4     __u32 dip;    //目的IP
5     __u32 sport;  //源端口
6     __u32 dport;  //目的端口
7     __u32 family; //协议
8 };
```

有了键类型之后，就可以使用 SEC 关键字来定义套接字映射了，如下所示：

 复制代码

```
1 #include <linux/bpf.h>
2
3 struct bpf_map_def SEC("maps") sock_ops_map = {
4     .type = BPF_MAP_TYPE_SOCKHASH,
5     .key_size = sizeof(struct sock_key),
6     .value_size = sizeof(int),
7     .max_entries = 65535,
```

```
8     .map_flags = 0,  
9 };
```

为了方便后续在 eBPF 程序中引用这两个数据结构，你可以把它们保存到一个头文件 `sockops.h` 中（你还可以在 [GitHub](#) 中找到完整的代码）。

更新套接字映射

套接字映射准备好之后，第二步就是在 `BPF_PROG_TYPE_SOCKET_OPS` 类型的 eBPF 程序中跟踪套接字事件，并把套接字信息保存到 `SOCKHASH` 映射中。

参考内核中 `BPF_PROG_TYPE_SOCKET_OPS` 程序类型的 [定义格式](#)，它的参数格式为 [struct bpf_sock_ops](#)：

```
1 #define BPF_PROG_TYPE(_id, _name, prog_ctx_type, kern_ctx_type)  
2  
3 BPF_PROG_TYPE(BPF_PROG_TYPE_SOCKET_OPS, sock_ops,  
4     struct bpf_sock_ops, struct bpf_sock_ops_kern)
```

[复制代码](#)

因此，你就可以使用如下的格式来定义这个 eBPF 程序：

```
1 SEC("sockops")  
2 int bpf_sockmap(struct bpf_sock_ops *skops)  
3 {  
4     // TODO: 添加套接字映射更新操作  
5 }
```

[复制代码](#)

在添加具体的套接字映射更新逻辑之前，还需要你先从 `struct bpf_sock_ops` 中获取作为键类型的五元组。参考内核中 [struct bpf_sock_ops](#) 的定义，如下的几个字段刚好可以满足我们的需要：

```
1 struct bpf_sock_ops {  
2     __u32 family;  
3     __u32 remote_ip4; /* Stored in network byte order */
```

[复制代码](#)

```
4  __u32 local_ip4; /* Stored in network byte order */
5  __u32 remote_port; /* Stored in network byte order */
6  __u32 local_port; /* stored in host byte order */
7  ...
8  }
```

因此，你就可以直接使用它们来定义映射中所需要的键。下面就是 `sock_key` 的定义方法，注意这里把 `local_port` 转换为了同其他字段一样的网络字节序：

[复制代码](#)

```
1 struct sock_key key = {
2     .dip = skops->remote_ip4,
3     .sip = skops->local_ip4,
4     .sport = bpf_htonl(skops->local_port),
5     .dport = skops->remote_port,
6     .family = skops->family,
7 };
```

有了键之后，还不能立刻就去更新套接字映射。这是因为 `BPF_PROG_TYPE_SOCK_OPS` 程序跟踪了所有类型的套接字操作，而我们只需要把新创建的套接字更新到映射中。

`struct bpf_sock_ops` 中包含的 `op` 字段可用于判断套接字操作类型，其定义格式可以参考 [这里的内核头文件](#)。内核头文件中已经为每种操作的具体含义加了详细的注释，对于新创建的连接，我们就可以使用以下两个状态（即主动连接和被动连接）作为判断条件：

[复制代码](#)

```
1 /* skip if it is not established op */
2 if (skops->op != BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB && skops->op != BPF_SOCK_
3     return BPF_OK;
4 }
```

到这里，说明套接字已经属于新创建的连接了，所以接下来就是调用 BPF 辅助函数去更新套接字映射，如下所示：

[复制代码](#)

```
1 bpf_sock_hash_update(skops, &sock_ops_map, &key, BPF_NOEXIST);
```

其中，BPF_NOEXIST 表示键不存在的时候才添加新元素。

再加上必要的头文件，完整的 eBPF 程序如下所示：

[复制代码](#)

```
1 #include <linux/bpf.h>
2 #include <bpf/bpf_endian.h>
3 #include <bpf/bpf_helpers.h>
4 #include <sys/socket.h>
5 #include "sockops.h"
6
7 SEC("sockops")
8 int bpf_sockmap(struct bpf_sock_ops *skops)
9 {
10     /* skip if the packet is not ipv4 */
11     if (skops->family != AF_INET)
12     {
13         return BPF_OK;
14     }
15
16     /* skip if it is not established op */
17     if (skops->op != BPF SOCK OPS PASSIVE_ESTABLISHED_CB && skops->op != BPF_S
18         return BPF_OK;
19     }
20
21     struct sock_key key = {
22         .dip = skops->remote_ip4,
23         .sip = skops->local_ip4,
24         /* convert to network byte order */
25         .sport = (bpf_htonl(skops->local_port)),
26         .dport = skops->remote_port,
27         .family = skops->family,
28     };
29
30     bpf_sock_hash_update(skops, &sock_ops_map, &key, BPF_NOEXIST);
31     return BPF_OK;
32 }
33
34 char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

把上述代码保存到 sockops.bpf.c 文件中（你还可以在 [GitHub](#) 中找到完整的代码），然后执行下面的命令，将其编译为 BPF 字节码：

[复制代码](#)

```
1 clang -g -O2 -target bpf -D__TARGET_ARCH_x86 -I/usr/include/x86_64-linux-gnu -
```

到这里，套接字更新的 eBPF 程序就准备好了，接下来我们来看看如何转发套接字。

套接字转发

第三步的套接字转发可以使用 `BPF_PROG_TYPE_SK_MSG` 类型的 eBPF 程序，捕获套接字中的发送数据包，并根据上述的套接字映射进行转发。根据内核头文件中的 [定义格式](#)，它的参数格式为 [struct sk_msg_md](#)。struct sk_msg_md 的定义格式如下所示，也已经包含了套接字映射所需的五元组信息：

[复制代码](#)

```
1 struct sk_msg_md {
2     ...
3     __u32 family;
4     __u32 remote_ip4; /* Stored in network byte order */
5     __u32 local_ip4; /* Stored in network byte order */
6     __u32 remote_port; /* Stored in network byte order */
7     __u32 local_port; /* stored in host byte order */
8     ...
9 };
```

了解清楚数据结构的定义格式之后，还需要你注意一点：`BPF_PROG_TYPE_SK_MSG` 跟 `BPF_PROG_TYPE_SOCKET_OPS` 属于不同的 eBPF 程序。虽然你可以把多个 eBPF 程序放入同一个源码文件，并编译到同一个字节码文件（即 文件名.o）中，但由于它们的加载和挂载格式都是不同的，我推荐你把不同的 eBPF 程序放入不同的文件中，这样管理起来更为方便。

因此，接下来创建一个新的文件（如 `sockredir.bpf.c`），用于保存 `BPF_PROG_TYPE_SK_MSG` 程序。添加如下的代码，就定义了一个名为 `bpf_redir` 的 eBPF 程序：

[复制代码](#)

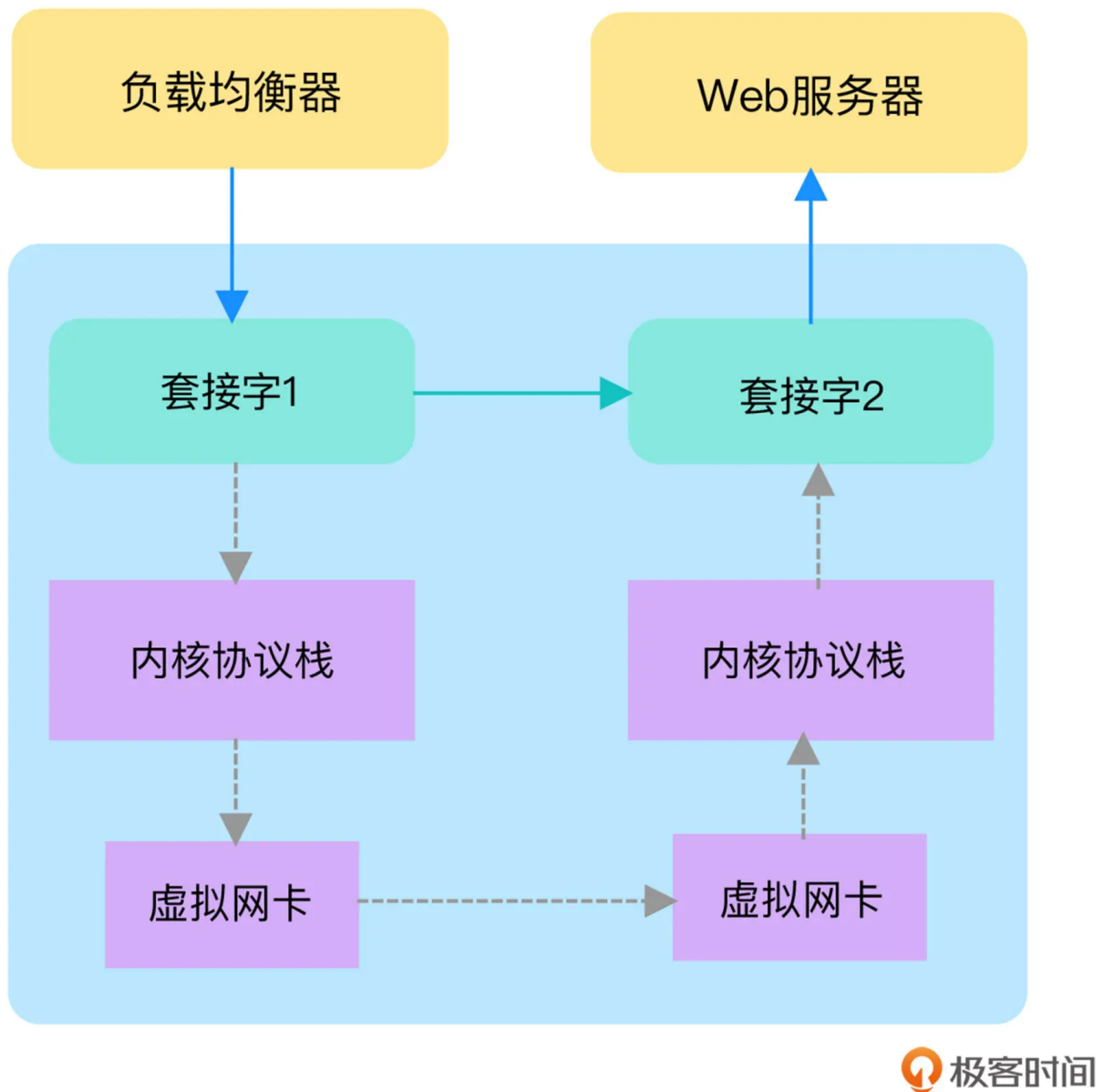
```
1 SEC("sk_msg")
2 int bpf_redir(struct sk_msg_md *msg)
3 {
4     //TODO: 添加套接字转发逻辑
5 }
```

在这个 eBPF 程序中，既然还要访问相同的套接字映射，也就需要从参数 `struct sk_msg_md` 中提取五元组信息，并存入套接字映射所需要的键 `struct sock_key` 中。如下所示，我们就定义了一个新的 `struct sock_key`（注意，这里同样需要把 `local_port` 转换为网络字节序）：

 复制代码

```
1 struct sock_key key = {
2     .sip = msg->remote_ip4,
3     .dip = msg->local_ip4,
4     .dport = bpf_htonl(msg->local_port),
5     .sport = msg->remote_port,
6     .family = msg->family,
7 };
```

需要你注意的是，这儿的源 IP 和源端口对应上述 eBPF 程序的目的 IP 和目的端口，也就是说，发送方向刚好是相反的。为什么是相反的呢？来看看下面这张图，原因就很清楚了：



图中，灰色箭头是套接字转发之前的网络流向，而绿色箭头则是套接字转发后的网络流向。从这张图中你可以发现：

在套接字转发之前，即便是在同一台机器的两个容器中，负载均衡器和 Web 服务器的两个套接字通信还是需要通过完整的内核协议栈进行处理的；

而在套接字转发之后，来自发送端套接字 1 的网络包在套接字层就交给了接收端的套接字 2，从而避免了额外的内核协议栈处理过程。

由于这两个套接字一个是发送，一个是接收，因而它们的方向是相反的，所以在构造转发套接字的键时，就需要把源和目的交换。

有了套接字映射所需要的键之后，最后还剩下添加套接字转发逻辑的步骤。参考 BPF 辅助函数文档（你可以执行 `man bpf-helpers` 查询），`bpf_msg_redirect_hash()` 正好跟我们的需求完全匹配。为了方便你理解，我把它的使用文档也贴一下：

[复制代码](#)

```
1 long bpf_msg_redirect_hash(struct sk_msg_buff *msg, struct bpf_map *map, void
2
3 Description
4 This helper is used in programs implementing policies at the socket level. I
5 message msg is allowed to pass (i.e. if the verdict eBPF program returns SK_
6 redirect it to the socket referenced by map (of type BPF_MAP_TYPE_SOCKHASH)
7 hash key. Both ingress and egress interfaces can be used for redirectio
8 BPF_F_INGRESS value in flags is used to make the distinction (ingress path i
9 lected if the flag is present, egress path otherwise). This is the only fla
10 ported for now.
11
12 Return SK_PASS on success, or SK_DROP on error.
```

概括来说，`bpf_msg_redirect_hash()` 的作用就是把当前套接字转发给套接字映射中的套接字。而参数 `key` 用于从套接字映射中查询待转发的套接字，`flags` 用于区分入口或出口路径。

根据每个参数的具体格式，你可以通过下面的方式进行套接字转发。注意，对于负载均衡的场景来说，只需要对入口路径进行处理，因而这儿设置了 `BPF_F_INGRESS`。

[复制代码](#)

```
1 bpf_msg_redirect_hash(msg, &sock_ops_map, &key, BPF_F_INGRESS);
```

再加上必要的头文件之后，完整的 eBPF 程序如下所示：

[复制代码](#)

```
1 #include <linux/bpf.h>
2 #include <bpf/bpf_endian.h>
3 #include <bpf/bpf_helpers.h>
4 #include <sys/socket.h>
5 #include "sockops.h"
6
7
8
9 SEC("sk_msg")
```

```
10 int bpf_redir(struct sk_msg_md *msg)
11 {
12     struct sock_key key = {
13         .sip = msg->remote_ip4,
14         .dip = msg->local_ip4,
15         .dport = bpf_htonl(msg->local_port),
16         .sport = msg->remote_port,
17         .family = msg->family,
18     };
19
20     bpf_msg_redirect_hash(msg, &sock_ops_map, &key, BPF_F_INGRESS);
21     return SK_PASS;
22 }
23
24 char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

把上述代码保存到文件 `sockredir.bpf.c` 中（你还可以在 [GitHub](#) 中找到完整的代码），然后执行下面的命令，将其编译为 BPF 字节码：

[复制代码](#)

```
1 clang -g -O2 -target bpf -D__TARGET_ARCH_x86 -I/usr/include/x86_64-linux-gnu -
```

加载 eBPF 程序

得到套接字映射更新和转发这两个 BPF 字节码之后，还需要把它们加载到内核之中，再挂载到特定的内核事件之后才会生效。在之前的案例中，我介绍的方法是利用 BCC、libbpf 等提供的库函数。今天，我再为你介绍另外一种方法，即通过命令行工具 `bpftool` 加载和挂载 eBPF 程序。

首先，对于 `sockops` 程序 `sockops.bpf.o` 来说，你可以执行下面的命令，将其加载到内核中：

[复制代码](#)

```
1 sudo bpftool prog load sockops.bpf.o /sys/fs/bpf/sockops type sockops pinmaps
```

这条命令将 `sockops.bpf.o` 中的 eBPF 程序和映射加载到内核中，并固定到 BPF 文件系统中。固定到 BPF 文件系统的好处是，即便 `bpftool` 命令已经执行结束，eBPF 程序还会继续在内核中运行，并且 eBPF 映射也会继续存在内核内存中。

加载成功后，你还可以执行 `bpftool prog show` 和 `bpftool map show` 命令确认它们的加载结果。执行成功后，你会看到类似下面的输出：

[复制代码](#)

```
1 $ sudo bpftool prog show name bpf_sockmap
2 1062: sock_ops name bpf_sockmap tag e37ef726a3a85a2e gpl
3   loaded_at 2022-02-04T13:07:28+0000 uid 0
4   xlated 256B jited 140B memlock 4096B map_ids 90
5   btf_id 234
6
7 $ sudo bpftool map show name sock_ops_map
8 90: sockhash name sock_ops_map flags 0x0
9   key 20B value 4B max_entries 65535 memlock 1572864B
```

BPF 字节码加载成功之后，其中的 eBPF 程序还不会自动运行，因为这时候它还没有与内核事件挂载。

对 sockops 程序来说，它支持挂载到 cgroups，从而对 cgroups 所拥有的所有进程生效，这跟我们案例的容器场景也是匹配的。

虽然 Docker 支持把新容器挂载到 cgroups 子系统中，但在案例开始的时候，我们并没有指定 cgroups 子系统。此时，Docker 会自动把所有容器都添加到系统 cgroups 子系统中。所以，对 sockops 程序来说，就可以把它挂载到系统 cgroups 中，从而对包括容器应用在内的所有进程生效。

你可以执行下面的 `mount` 命令，查询当前系统的 cgroups 的挂载路径：

[复制代码](#)

```
1 $ mount | grep cgroup
2 cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,relatime,nsdele
```

通常情况下，主流的发行版都会把 cgroups 挂载到 `/sys/fs/cgroup` 路径下。接着，再执行下面的 `bpftool cgroup attach` 命令，把 sockops 程序挂载到 cgroups 路径中：

[复制代码](#)

```
1 sudo bpftool cgroup attach /sys/fs/cgroup/ sock_ops pinned /sys/fs/bpf/sockops
```

到这里，sockops 程序的加载和挂载就完成了。

接下来，再执行下面的命令，加载并挂载 sk_msg 程序 sockredir.bpf.o：

[复制代码](#)

```
1 sudo bpftool prog load sockredir.bpf.o /sys/fs/bpf/sockredir type sk_msg map n
2 sudo bpftool prog attach pinned /sys/fs/bpf/sockredir msg_verdict pinned /sys/
```

从这两条命令中你可以看到，sk_msg 程序的加载和挂载过程跟 sockops 程序是类似的，区别只在于它们的程序类型和挂载类型不同：

sockops 程序的类型是 sock_ops，sk_msg 程序的类型是 sk_msg；

sockops 程序的挂载类型是 cgroup（对应 bpftool cgroup attach 命令），

sk_msg 程序的挂载类型是 msg_verdict（对应 bpftool prog attach 命令）。

由于 sk_msg 程序需要访问 sockops 程序创建的套接字映射，所以上述命令通过 BPF 文件系统路径 /sys/fs/bpf/sock_ops_map 对套接字映射进行了绑定。

到这里，两个 eBPF 程序的加载和挂载就都完成了。

那么，它们是不是真的可以提升网络转发的性能呢？回想一下 Nginx 负载均衡的测试步骤，我们使用相同的方法再做个性能测试，就可以知道了。

性能测试

执行下面的命令进入 client 容器终端，并在容器终端中执行 wrk 命令：

[复制代码](#)

```
1 docker exec -it client sh
2 / # wrk -c100 "http://172.17.0.5"
```

稍等一会，你会看到如下的输出（在你的环境下可能看到不同数值，具体的性能指标取决于运行环境和配置）：

[复制代码](#)

```
1 Running 10s test @ http://172.17.0.5
2 2 threads and 100 connections
3 Thread Stats Avg Stdev Max +/- Stdev
4 Latency 6.88ms 4.71ms 46.08ms 70.77%
5 Req/Sec 7.70k 548.11 9.10k 66.50%
6 153466 requests in 10.03s, 24.15MB read
7 Requests/sec: 15300.71
8 Transfer/sec: 2.41MB
```

你可以看到，新的平均每秒请求数是 15300，相比优化之前的 13798 提升了 10.8%；而每个线程的平均延迟 6.88ms 也比之前的 7.53ms 降低了 8.6%。这说明，eBPF 真的优化了负载均衡器的转发性能，这跟我们一开始的猜想是一致的。

案例清理

案例的最后，不要忘记清理今天所加载的 eBPF 程序以及容器环境。

对于 eBPF 程序来说，清理过程需要卸载（detach）和删除（unload）两个步骤。执行下面的命令，就可以卸载和删除 skops 和 sk_msg 这两个程序：

[复制代码](#)

```
1 # cleanup skops prog and sock_ops_map
2 sudo bpftool cgroup detach /sys/fs/cgroup/ sock_ops name bpf_sockmap
3 sudo rm -f /sys/fs/bpf/sockops /sys/fs/bpf/sock_ops_map
4
5 # cleanup sk_msg prog
6 sudo bpftool prog detach pinned /sys/fs/bpf/sockredir msg_verdict pinned /sys/
7 sudo rm -f /sys/fs/bpf/sockredir
```

你可能已经注意到了，与 attach 相对应的清理操作为 detach，但 bpftool 并没有一个与 load 相对应的 unload 子命令。这是因为，eBPF 程序和映射都是与 BPF 文件系统绑定的，文件删除后，它们引用计数降为 0，就会被系统自动清理了，所以删除过程只需要把 BPF 文件系统文件删除即可。

而对于容器的清理就更容易了，只需要执行下面的 `docker rm` 命令即可：

```
1 docker rm -f http1 http2 client nginx
```

[复制代码](#)

小结

今天，我带你搭建了一个最简单的负载均衡程序，并借助套接字 eBPF 程序对它的性能进行了优化。

借助 `sockops` 和 `sk_msg` 等套接字 eBPF 程序，你可以在内核态中直接将网络包转发给目的应用的套接字，跳过复杂的内核协议栈，从而提升网络转发的性能。

在需要加载和挂载 eBPF 程序或映射时，除了可以利用 `BCC`、`libbpf` 等提供的库函数之外，你还可以使用 `bpftool` 这个工具来实现。由于 eBPF 程序及相关的工具还在快速进化中，在碰到不确定的疑问时，我推荐你参考跟当前内核版本匹配的内核头文件定义、man 手册等，去查询关于它们的详细文档，而不要单纯依赖于网络搜索。

当然，对于网络优化来说，除了套接字 eBPF 程序之外，XDP 程序和 TC 程序也是最常用的性能优化方法，我将在下一讲中为你介绍 XDP 程序的详细使用方法。

思考题

最后的思考题环节，我们来探讨关于排查套接字映射的问题。

由于今天的案例中没有添加日志，所以要想观察 eBPF 程序的运行状态，只能通过套接字映射，而 `bpftool map dump` 命令则提供了查看套接字映射内容的功能。

在终端中执行下面的 `nc` 命令，新建一个到 Nginx 容器的连接：

```
1 nc 172.17.0.5 80
```

[复制代码](#)

然后打开一个新终端，执行下面的 `bpftool map dump` 命令：

 复制代码

```
1 sudo bpftool map dump name sock_ops_map
```

接着，你就会看到类似下面的输出：

 复制代码

```
1 key:
2 ac 11 00 01 ac 11 00 05 00 00 be 12 00 00 00 50
3 02 00 00 00
4 value:
5 No space left on device
6 key:
7 ac 11 00 05 ac 11 00 01 00 00 00 50 00 00 be 12
8 02 00 00 00
9 value:
10 No space left on device
```

由于 value 对应的是套接字文件描述符，bpftool 无法显示它的内容，所以你可以忽略 No space left on device 的错误（实际上这个错误信息是不准确的，新版本中已经修复了错误信息）。

而对于 key 来说，则是一长串的十六进制数值。这里我想请你思考两个问题：

1. 这些十六进制数值是什么意思？它们是如何跟 eBPF 程序中定义的 struct sock_key 关联的？
2. 你可以尝试把这些十六进制数值转换成 struct sock_key 数据结构，并计算出每个属性的值吗？

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta 订阅超级会员，你将得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 容器安全：如何使用 eBPF 增强容器安全？

精选留言 (2)



maosd199554

2022-02-12

老师，这种只适合LB跟服务部署同一个服务器吧？

作者回复：嗯嗯，是的



莫名 

2022-02-11

key:

```
ac 11 00 05 ac 11 00 01 00 00 00 50 00 00 be 12
02 00 00 00
```

以该 key 为例，struct sock_key 共 5 个 u32 类型的字段，每个字段四个字节，其中：...
展开

作者回复：解析的非常正确。

对于你的疑问，其实你到客户端容器中执行相同的nc命令，再观察一下映射中的内容，答案自然就有了。这儿看到172.17.0.1是因为思考题中的nc命令是在主机终端中执行的。

